

ERROR DETECTION IN STORAGE DATA

BACKGROUND OF THE INVENTION

1. Field of the Invention

- 5 [0001] The present invention relates to a method, system, and an article of manufacture for implementing an error detection scheme in storage data.

2. Description of the Related Art

- 10 [0002] A block addressable storage device is typically comprised of one or more disks, such as flexible disks, rigid disks, optical discs, and stores data in addressable groups referred to as blocks. The number of bytes of data contained in a single block is called the block length or block size. While the block length can be any number of bytes, storage device manufacturers often format the storage devices into blocks with a block length of 512 bytes. The storage devices can be reformatted into blocks of a different block length.
- 15 Application programs that read and write data to the storage devices need assurance that data integrity is maintained as data is transferred between the storage device and application program.

- [0003] Prior art storage devices include techniques for assuring data integrity. For instance, storage device controllers often utilize an error correcting code (ECC) algorithm
- 20 to detect and possibly correct hardware related failures within the storage device. In addition to hardware errors, data integrity may be compromised by transport errors that occur during data transmission via Small Computer System Interface (SCSI) cables, storage adapter cards and storage device drivers. Failure to detect the transport errors, as well as disk error allows corrupt data to propagate. Undetected transport errors that occur
- 25 within data are referred to as "silent data corruption." Silent data corruption occurs when the application program retrieves data from the storage system (i.e. a disk read request) that is stale, altered or lost without being detected or corrected. Stale data is data that was written at an earlier time and is incorrectly returned in place of the more recent (lost) data. Altered data is data that is present but corrupted or changed and no longer correctly

represents the original data. Finally, lost data is data that is lost and no longer available. The presence of such errors is of substantial concern for critical applications where the impact of undetected errors can be catastrophic.

5 [0004] In prior art, checksums have been used to detect errors in data. The checksum of a group of data items is stored or transmitted with the group of data items. The checksum value is calculated by treating the data items as numeric values. Checksums are widely used in network protocols, where a checksum generated from the bits of a message accompanies the message during transmission. For instance, many checksum algorithms perform an XOR of the bits in the message to generate the checksum. The receiving
10 station then applies the same checksum algorithm (e.g. XOR) to the message and checks to make sure that the computed numerical value is the same as the checksum within the transmission. In view of the prevalence of silent data corruption, there is a need in the art to provide an improved checksum based technique to detect silent data corruption.

15 SUMMARY OF THE PREFERRED EMBODIMENTS

[0005] Provided are methods, systems and programs to detect errors while accessing a storage device. A host system retrieves one block from a plurality of blocks, wherein a pattern has been written into the plurality of blocks during initialization of the blocks in a storage device. The host system determines whether the retrieved block includes the
20 pattern and also determines whether a checksum computed from the retrieved block is different than the value in a checksum field of the retrieved block. The host system generates an error when data from the retrieved block from the plurality of blocks computes to a checksum that is different from the value contained within the checksum field for the retrieved block, and the retrieved block does not contain the initialization
25 pattern.

[0006] Still further, the pattern written into the plurality of blocks while initializing data in the storage device is written during a format operation on the storage device, wherein no checksum value is written to the checksum field during the format operation.

[0007] Further implementations provide a system, method and article of manufacture for formatting a storage device into a plurality of blocks of a block size. A set of bits is reserved within each of the plurality of blocks, to store a pattern. A checksum field is reserved within each of the plurality of blocks, to store a checksum. The storage device is formatted into the plurality of blocks. During the formatting, the pattern is written into the set of bits within each of the plurality of blocks, wherein no checksum value is written to the checksum field during the formatting.

[0008] In one implementation, the formatting comprises executing of a SCSI FORMAT UNIT command. In further implementations the SCSI FORMAT UNIT command is configured by a host system and executed by the storage device. The implementations provide a checksum based technique that can detect errors.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] Referring now to the drawings in which like reference numbers represent corresponding parts throughout:

FIG. 1a illustrates the blocks of a storage device in accordance with implementations of the invention;

FIG. 1b illustrates the checksum field within a block of a storage device in accordance with implementations of the invention;

FIG. 1c illustrates a disk label within a block of a storage device in accordance with implementations of the invention;

FIG. 2 illustrates a block diagram of a first computing environment in which certain aspects of the invention are implemented;

FIG. 3 illustrates logic to write an initialization pattern when a host formats a disk in accordance with implementations of the invention;

FIG. 4 illustrates the use of SCSI FORMAT UNIT command in accordance with implementations of the invention;

FIG. 5 illustrates logic to write to a storage device in accordance with implementations of the invention; and

FIG. 6 illustrates logic to read from a storage device in accordance with implementations of the invention.

FIG. 7 illustrates the block diagram of a second computing environment in which certain implementations of the invention can be implemented.

5

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0010] In the following description, reference is made to the accompanying drawings which form a part hereof and which illustrate several implementations. It is understood that other implementations may be utilized and structural and operational changes may be made without departing from the scope of the present implementations.

[0011] Although checksums have been used within storage devices, existing checksum features within storage devices have limitations. For instance, checksums implemented within the storage device alone cannot detect silent data corruption which occurs between the storage device and the host applications. The copending and commonly assigned patent application entitled "End-to-End Disk Data Checksumming," having U.S. Patent Application No. 09/917,315 and filed on July 27, 2001, discloses an error detection scheme implemented in a host system to detect errors, such as transmission errors, resulting from silent data corruption. While writing data to a block of a storage device, the host calculates a checksum corresponding to the data and writes the checksum into the block of storage device. Subsequently, while reading the data from the block of the storage device, the host compares a checksum calculated from the read data to the previously written checksum. If the calculated checksum does not match the previously written checksum the host generates a checksum error.

[0012] To utilize the error detection scheme described in the patent application entitled "End-to-End Disk Data Checksumming," a host may write a checksum to every block of the storage device after formatting the storage device in order to avoid a checksum error during the first read after the disk is formatted. The error detection scheme described in the patent application entitled "End-to-End Disk Data Checksumming" does not allow the host to detect silent data corruption without having to write the checksum to every block

of the storage device after formatting the storage device. The process of writing the checksum to each block after formatting the storage device may be time consuming. For this reason, described implementations provide techniques to initialize the disk for error detection, such as the error detection schemes disclosed in "End-to-End Disk Data

5 Checksumming."

[0013] FIG. 1a illustrates a storage device 80 formatted into a plurality of n blocks, 1st block 100₁, 2nd block 100₂, ith block 100_i, nth block 100_n. The number of blocks in the storage device 80 can exceed several million or more when the storage device 80 has a large capacity. The storage device 80 may comprise any storage device known in the art, such as a single disk drive, a Direct Access Storage Device (DASD), Just a Bunch of
10 Disks (JBOD), a Redundant Array of Independent Disks (RAID), a tape library, an optical library, and so on. Manufacturers of the storage device 80 often preformat the storage device 80 into blocks of block length 512 bytes when shipping the storage device 80. However, it is possible to reformat the storage device 80 into blocks of any size.

15 [0014] FIG 1b illustrates the ith block 100_i contained within the storage device 80. The block 100_i contains data field 110_i and a checksum field 120_i. The checksum field 120_i stores a checksum corresponding to data in data field 110_i. In one implementation the checksum field 120_i is 16 bytes, and the data is 512 bytes giving a total of 528 bytes for the block length of each block. Each of the blocks 100₁, 100₂,...100_i, ...,100_n has the
20 checksum field 120_i and the data field 110_i. In certain implementations, the checksum is comprised of a 32-bit Exclusive-Or (XOR) checksum. However, the checksum can be calculated using any checksumming technique known in the art. The checksum for each block is kept with the data in the same block rather than in a separate location on the storage device 80. In alternative implementations, the checksum for each block can be
25 kept in a separate location on the storage device 80 or on a different storage device.

[0015] FIG. 1c illustrates the disk label 84 within the first block 100₁ of a storage device in accordance with implementations of the invention. A disk label 84 is part of the data field 110₁ on the first block 100₁ of the storage device 80, and the disk label 84

identifies the storage device 80. The disk label 84 also contains a checksumming flag 85. The checksumming flag 85 indicates whether the storage device 80 contains checksums.

[0016] FIG. 2 illustrates a computing environment for the implementation. A host system 2 includes an operating system 4 (the operating system includes device drivers) and is capable of executing multiple application programs 6a, 6b, 6c. The host 2 may comprise any computational device known in the art including a server class machine, a mainframe, a desktop computer, a laptop computer, a hand held computer, a telephony device. The operating system 4 may comprise any operating system known in the art capable of concurrently executing multiple application programs 6a, 6b, 6c and concurrently generating I/O requests. Although multiple applications can be executed on the host 2, only three application programs 6a, 6b, 6c are shown for illustration purposes. The applications can be any applications known in the art. The host system 2 further includes a target driver 10, two storage device drivers 24 and 26, and storage adapter cards 12 and 14. A format utility 30 is also present within host system 2. The format utility 30 can be used by the host system 2 to format storage devices (e.g. the storage device 80, or storage device 82). Alternatively, the code for the format utility 30 can be stored in a separate storage unit or run from a command prompt (e.g. UNIX shell prompt) manually from the host system 2, or can be run from any other system connected to the storage devices 80 and 82. Although two storage devices 80, 82 are shown, the host system 2 may communicate with only one or more than two storage devices.

[0017] The application programs 6a, 6b, 6c generate (I/O) requests to the two storage devices 80 and 82 (there may be additional storage devices), where the data files used by the application programs 6a, 6b, 6c are stored. In certain implementations, to coordinate the I/O process, all I/O requests are transferred from the application programs 6a, 6b, 6c to a single target driver 10 for communicating with multiple storage device drivers 24 and 26. The target driver 10 includes code to process, generate and verify checksum values. If an I/O request is for storage device 80, the target driver 10 determines which blocks within storage device 80 should be accessed to read/write the data used by the applications 6a, 6b, 6c. In addition, the target driver 10 adds a checksum to the data

blocks on the write function and subtracts the checksum on the read function, which will be explained in greater detail with respect to FIGs. 5 and 6. The target driver 10 can communicate with the storage devices 80, 82 through the storage device drivers 24, 26 and the storage adapter cards 12, 14 in, for example, any of a number of ways known to those of ordinary skill in the art.

[0018] The target driver 10 generates generic device commands in response to I/O requests from the applications 6a, 6b, 6c. The generic device commands are converted to device specific commands by the storage drivers 24 and 26. In certain implementations, the target driver 10 handles all the checksumming and error detection operations, thereby avoiding the need to modify the storage device drivers 24 and 26, which are typically supplied by the manufacturers of the corresponding storage devices. However, in alternative implementations, operations described as performed by the target driver can be performed by the storage device drivers.

[0019] FIG. 3 illustrates logic implemented in the target driver 10 to format and write initialization patterns in a single pass through the storage device 80 when the host 2 formats the storage device 80 in accordance with an implementation of the invention. Control begins with the host 2 reading (at block 400) the information related to the storage device 80 where the storage device 80 has been formatted into 512 byte blocks. The information related to the storage device 80 may be kept either within the storage device 80 or in various configuration files within host 2. The various configuration files may be created when the storage device 80 and the storage adapter card 12 are attached to the host 2, and the host 2 installs the storage device driver 24. At block 405, the host 2 determines an initialization pattern to be written in every block of the storage device 80. The initialization pattern is stored in a sequence of bits. A FOR loop begins at block 410 and extends to block 420. At block 410, the block number i is incremented from 1 to the number of blocks, n , at every loop of the FOR loop. At the conclusion of block 410, the host executes block 415a and 415b in a single operation. Alternatively the host may execute block 415a and then immediately execute block 415b. At block 415a, host 2 formats block i into 528 bytes. At block 415b, the host 2 writes the initialization pattern

5 onto block *i*. At the conclusion of blocks 415a and 415b, control passes (at block 420) to the end of the FOR loop and control passes back to block 410. After the host 2 formats and writes the initialization pattern on block *n* of the storage device 80, control passes to block 425, where the host 2 modifies the disk label 84 by writing information pertaining to the initialization pattern on storage device 80. The host 2 also sets the checksumming flag 85 on disk label 84, to indicate that the disk may operate as a checksummed disk. Host 2 may modify the process of FIG. 3 for different storage devices and storage device drivers. The modifications may depend on the particular specifications of the storage device 80. The particular specifications may depend on the industry standard the storage device 80 supports, or may be proprietary to storage device 80.

10 [0020] FIG. 4 illustrates various data structures associated with the FORMAT UNIT command 450 specified in the Small Computer System Interface 3 (SCSI-3) specification in accordance with implementations of the invention. The FORMAT UNIT command 450 includes data structures that are used to provide an initialization pattern for use when formatting the blocks. Certain implementations use the FORMAT UNIT command 450 of the SCSI-3 specification to format and write initialization patterns in a single pass through the storage device 80 (as described in FIG. 3). Further details of the FORMAT UNIT command 450 are described in the publication entitled "Information Technology - SCSI-3 Block Commands" by American National Standards Institute, Inc. (revision 8c, published on 13 November 1997; also available as technical report/standard NCITS 306:1998), which publication is incorporated herein by reference in its entirety.

20 [0021] FIG. 4 shows the FORMAT UNIT COMMAND 450 having the FMTDATA 452 field, and other fields. In certain implementations, the host 2 sets the FMTDATA 452 field to one. When the FMTDATA field is set to one, any program that interprets the FORMAT UNIT COMMAND looks for the FORMAT UNIT parameter list 454. The FORMAT UNIT parameter list 454 starts with the DEFECT LIST HEADER 456. The host 2 sets the IP 462 field to one in the DEFECT LIST HEADER 456. When the IP 462 field is set to one, any program that interprets the FORMAT UNIT COMMAND 450 looks for the INITIALIZATION PATTERN DESCRIPTOR field 458. In the

INITIALIZATION PATTERN DESCRIPTOR field 458, the host 2 sets the IP MODIFIER 466 field to be zero, the INITIALIZATION PATTERN LENGTH 470 field to the number of bytes in the initialization pattern, the INITIALIZATION PATTERN 472 field to a pattern, and the PATTERN TYPE 468 field to be one to fill the pattern

- 5 repeatedly within the number of bytes in the initialization pattern. For example, if the pattern is hexadecimal 5A3C and there are eight bytes in the initialization pattern, then the initialization pattern is hexadecimal 5A3C5A3C5A3C5A3C.

- [0022] FIG. 5 illustrates logic in the host 2 to write to the storage device 80 in accordance with implementations of the invention. Control begins at block 600 when the
- 10 target driver 10 receives a write request from one application 6a, 6b, 6c. Since most storage device drivers manipulate data in 512 byte sized blocks, the write request is processed in the target driver 10 in data blocks of 512-bytes. Before target driver 10 performs the write function, the target driver 10 first checks the checksumming flag 85 on the disk label 84 and determines (at block 602) whether checksumming is enabled for the
- 15 storage device 80. If the storage device 80 does not have checksumming enabled, then the checksum operation cannot be performed and the write request is sent (at block 606) to the storage device 80 via the storage adapter card 12 and the storage device driver 24 without using the checksum algorithm. However, if the storage device 80 has checksumming enabled, the target driver 10 (at block 604) allocates memory and
- 20 computes a checksum corresponding to the 512 byte data blocks of the write request. The checksum is then stored with the write data to increase the data size of the write data block up to 528-bytes. At block 606, the 528 byte data block (with the checksum included) is then sent to the storage device driver 24, which communicates directly with the storage adapter card 12. The storage device driver 24 instructs (at block 608) the
- 25 storage adapter card 12 to write the data block with the checksum onto storage device 80 at the location determined by the target driver 10 in a manner known in the art.

[0023] FIG. 6 illustrates logic in host 2 to read from a storage device in accordance with the implementation. Control begins at block 700 when the target driver 10 receives a read request from one application 6a, 6b, 6c. Upon identifying the read request, the target

driver 10 locates the address where the data blocks are stored on the storage device 80.

The target driver 10 then sends the location of the requested data blocks to the storage device driver 24, and the storage device driver 24 retrieves (at block 702) the data blocks from the storage device 80 through the storage adapter card 12 in a manner known in the art.

5 art.

[0024] At block 704, the target driver 10 determines whether the retrieved data block has incorporated the checksum feature by reading the checksumming flag 85 on the disk label 84. If the checksumming flag 85 indicates that checksumming is not enabled for this device, then the target driver 10 will understand that the checksum feature was not added during the write function, and the checksum algorithm will not be used. In such case, the retrieved data blocks are sent (at block 714) to the application program 6a, 6b, or 6c in a manner known in the art. However, if checksum is enabled, the target driver 10 will use the checksum algorithm. Thus, in certain implementations the same target driver 10 can be used in a storage device 80 using 512-byte unchecksummed disk blocks or 528-
10 byte checksummed disk blocks. At block 706, the target driver 10 calculates the checksum associated with the retrieved data block. At block 708, a compare function is performed between the calculated checksum and the checksum contained within checksum field 120_i of the retrieved data block. The compare operator will return a value of "true" only if both values are the same. If a "true" value is returned, then the target
15 driver 10 (at block 712) strips away the checksum from the read data to decrease the data size of the write data block from 528-bytes to 512-bytes.

[0025] At block 714, the data blocks are sent to the application program 6a, 6b, or 6c in the format that the application program 6a, 6b, or 6c can read (i.e. 512-byte disk blocks). If a "false" value is returned at block 708, the target driver 10 determines whether the
20 block of data was read from a block that has not been written to before. In block 710, the target driver 10 determines whether the initialization pattern is present on the block of data 100_i. If the initialization pattern occupies all the bytes of the data field 110_i on the block of data 100_i, then control proceeds to block 712 to remove the checksum and return the data, which in this case is just the initialization pattern. In alternative implementations

where the initialization pattern is kept on some of the bytes of the data field 110_i of the block of data 100_i, the initialization pattern may be removed before returning the data. In other alternative implementations where the initialization pattern is kept in the checksum field 120_i, the checksum field 120_i may be removed before returning the data. If in block 5 710, the initialization pattern is absent on the block of data 100_i, then the target driver 10 concludes that an incorrect data block was returned or that the data was corrupt. At block 718, an error message is sent to the application program 6a, 6b, or 6c, notifying that an error was detected during the read request.

10 [0026] Including the checksum program at the host or driver level, versus within the disk drive or storage device enclosure, allows detection of silent data corruption that occurs between the disk drive and the host. As stated, silent data corruption may result from transport errors occurring in the SCSI cables, storage driver adapters, storage device drivers, etc. By placing the checksum routine at the host or driver level, silent data corruption occurring upstream from the disk drive is detected. In addition, locating the 15 checksum routine at the host or driver level implements the checksum independent of the hardware. No additional hardware is required to perform the checksum function. Instead, a software update can be performed to an existing host system to install an updated target driver containing the checksum program. Furthermore, keeping the checksum function at the host or driver level allows the checksum to remain functionally transparent to users, 20 operating without affecting existing applications on the host system or requiring updates or modifications to host system applications.

[0027] FIG. 7 illustrates another implementation where in addition to the host 2 there is another host 2a and a server 2b. The server 2b can be any computational device known in the art. The hosts 2 and 2a and the server 2b are connected to the storage devices 80 and 25 82 via a network 3. The server 2b formats and writes the initialization pattern on the storage devices 80 and 82. Subsequently, the host systems 2 and 2a and optionally the server 2b, read from and write to the storage devices 80 and 82. FIG. 7 illustrates the situation where one computational device formats the storage devices and subsequently a plurality of computational devices perform read and write operations with respect to the

storage devices 80 and 82. The servers 2, 2a, 2b all include the target driver 10 and perform the operations described in FIGs. 5 and 6. In one implementation, the storage devices 80 and 82 may be part of a storage area network geographically distant from the location of the host systems 2 and 2a. Similarly, the server 2b may be geographically distant from the location of the host systems 2 and 2a.

Additional Implementation Details

[0028] The described error detection techniques may be implemented as a method, apparatus or article of manufacture using standard programming and/or engineering techniques to produce software, firmware, hardware, or any combination thereof. The term “article of manufacture” as used herein refers to code or logic implemented in hardware logic (e.g., an integrated circuit chip, Programmable Gate Array (PGA), Application Specific Integrated Circuit (ASIC), etc.) or a computer readable medium such as, magnetic storage medium (e.g. hard disk drives, floppy disks, tape), optical storage (e.g., CD-ROMs, optical disks, etc.), volatile and non-volatile memory devices (e.g., EEPROMs, ROMs, PROMs, RAMs, DRAMs, SRAMs, firmware, programmable logic, etc.). Code in the computer readable medium is accessed and executed by a processor. The code in which implementations are made may further be accessible through a transmission media or from a file server over a network. In such cases, the article of manufacture in which the code is implemented may comprise a transmission media, such as a network transmission line, wireless transmission media, signals propagating through space, radio waves, infrared signals, etc. Of course, those skilled in the art will recognize that many modifications may be made to this configuration without departing from the scope of the implementations, and that the article of manufacture may comprise any information bearing medium known in the art.

[0029] The described implementations show the host both formatting the storage devices as well as reading and writing to the storage devices. The described implementations provided a technique for managing the flow of I/Os to a device driver for a storage device. A 32-bit XOR algorithm was used for checksumming. Alternatively,

other algorithms can be used for checksumming which will compare the checksum of the retrieved data with the requested data, such as the Fletcher32-p algorithm. In addition, the checksum was described with a certain number of bytes of information.

Alternatively, the storage device can be reformatted to any other disk block length to

- 5 increase or decrease the disk block length to accommodate different checksum sizes and data blocks. For example, a 520-byte block size maybe used instead of a 528-byte data block. In addition, in the described implementations, the determination of whether the checksum feature was enabled on the storage device was made by checking the checksumming flag on the disk label. Additionally, a determination of whether the
- 10 checksum feature is enabled can be performed by checking the size of the data blocks in the storage device without the use of the checksumming flag.

[0030] In the described implementations, the checksumming feature was performed by the target driver. Alternatively, a checksumming driver could just be layered on top of any of the current I/O driver stacks or implemented in another layer of the I/O subsystem.

- 15 In addition, the code of the target driver 10 is described as excluding the device driver code for the storage device. Alternatively, the target driver may include the code for one or more device drivers.

[0031] The preferred logic of FIGs. 3, 5 and 6 described specific operations occurring in a particular order. Further, the steps may be performed in parallel as well as sequentially.

- 20 In alternative embodiments, certain of the logic operations may be performed in a different order, modified or removed and still implement preferred embodiments of the present invention. Moreover, steps may be added to the above described logic and still conform to the preferred embodiments.

- [0032] Therefore, the foregoing description of the implementations has been presented
- 25 for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto. The above specification, examples and data provide a complete description of the manufacture

[illegible]